Trenz Pruca
15. Dezember 2014

Afrikanische Tierwelt

Lorem ipsum dolor sit amet

Strategy

Ein Strategy Pattern kapselt einen Algorithmus in einer Klasse.
Wir benutzen dieses Entwurfsmuster, wenn wir einen Algorithmus unabhängig von nutzenden Clients austauschen.

„Strategie-Objekte werden ähnlich wie Klassenbibliotheken verwendet. Im Gegensatz dazu handelt es sich jedoch nicht um externe Programmteile, die als ein Toolkit genutzt werden können, sondern um integrale Bestandteile des eigentlichen Programms, die deshalb als eigene Objekte definiert wurden, damit sie durch andere Algorithmen ausgetauscht werden können." [1]

„Wir verwenden das Strategy Pattern wenn wir:

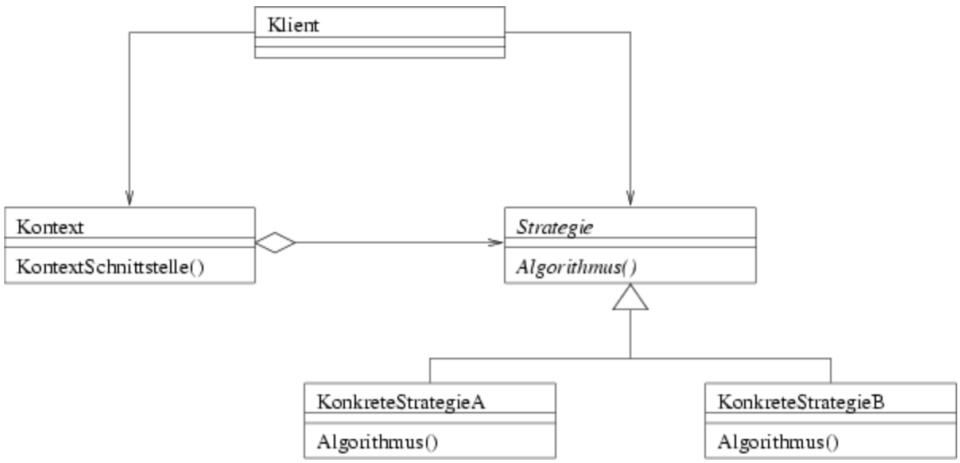
- viele verwandte Klassen sich nur in ihrem Verhalten unterscheiden.
- unterschiedliche (austauschbare) Varianten eines Algorithmus benötigt werden.
- Daten innerhalb eines Algorithmus vor Klienten verborgen werden sollen.
- verschiedene Verhaltensweisen innerhalb einer Klasse fest integriert sind (meist über Mehrfachverzweigungen) aber
- die verwendeten Algorithmen wiederverwendet werden sollen bzw.
- die Klasse flexibler gestaltet werden soll.

Daraus bilden sich jedoch auch Nachteile:

- Klienten müssen die unterschiedlichen Strategien kennen, um zwischen ihnen auswählen und den Kontext initialisieren zu können.
- Gegenüber der Implementierung der Algorithmen im Kontext entsteht hier ein zusätzlicher Kommunikationsaufwand zwischen Strategie und Kontext.
- Die Anzahl von Objekten wird erhöht." [1]

„Die Klasse Strategie definiert nur eine Schnittstelle (interface) für alle unterstützten Algorithmen. Die Implementierung der eigentlichen Algorithmen finden sich erst in den Ableitungen wieder (konkrete Strategie).

Der Kontext hält eine Member-Variable der Schnittstelle Strategie, die mit einer Referenz auf das gewünschte Strategieobjekt belegt ist. Auf diese Weise wird der konkrete Algorithmus über die Schnittstelle eingebunden und kann bei Bedarf selbst zur Laufzeit dynamisch aktualisiert werden.“ [1]



Strategy pattern in action

```python
class Context(object):
    def __init__(self, strategy):
        self.strategy = strategy

    def execute(self, a, b):
        return self.strategy.execute(a, b)

# Fuegt Faehigkeiten hinzu
class AddStrategy(object):
    def execute(self, a, b):
        return a + b


class SubtractStrategy(object):
    def execute(self, a, b):
        return a - b


class MultiStrategy(object):
    def execute(self, a, b):
        return a*b

# Neue Strategie +
context = Context(AddStrategy())
print('4 + 3 =', context.execute(4, 3))
# 4 + 3 = 7

context.strategy = SubtractStrategy()
print('4 - 3 =', context.execute(4, 3))
# 4 - 3 = 1

context.strategy = MultiStrategy()
print('2 * 2 =', context.execute(2, 2))
# 2 * 2 = 4
```

Abbildung [1] ergänztes Beispiel

Decorator

Das Entwurfsmuster „Decorator“, gibt uns die Möglichkeit Fähigkeiten für ein Objekt dynamisch zu erweitern.
„Anstatt Unterklassen zu bilden und eine Klasse damit um Fähigkeiten bzw. Verhalten zu erweitern, lässt sich mit dem Einsatz des Decorator Patterns die Erzeugung von Unterklassen vermeiden.

Das Decorator Pattern kann eingesetzt werden, wenn:

- … einzelnen Objekten zusätzliches Verhalten und zusätzliche Eigenschaften hinzugefügt werden sollen, ohne andere Objekte zu beeinflussen.
- … das Verhalten und die Eigenschaften auch wieder entfernt werden sollen.
- … die Bildung von Unterklassen zu einem zu komplexen und großen Klassenmodell führen würde.
- … die Bildung von Unterklassen nicht möglich ist, da z.B. Klassen nicht sinnvoll abgeleitet werden können.
- … die Implementation des zusätzlichen Verhaltens und der zusätzlichen Eigenschaften in der Ursprungsklasse zu komplex und zu oft zu überflüssigem Code führen würde. „[5]

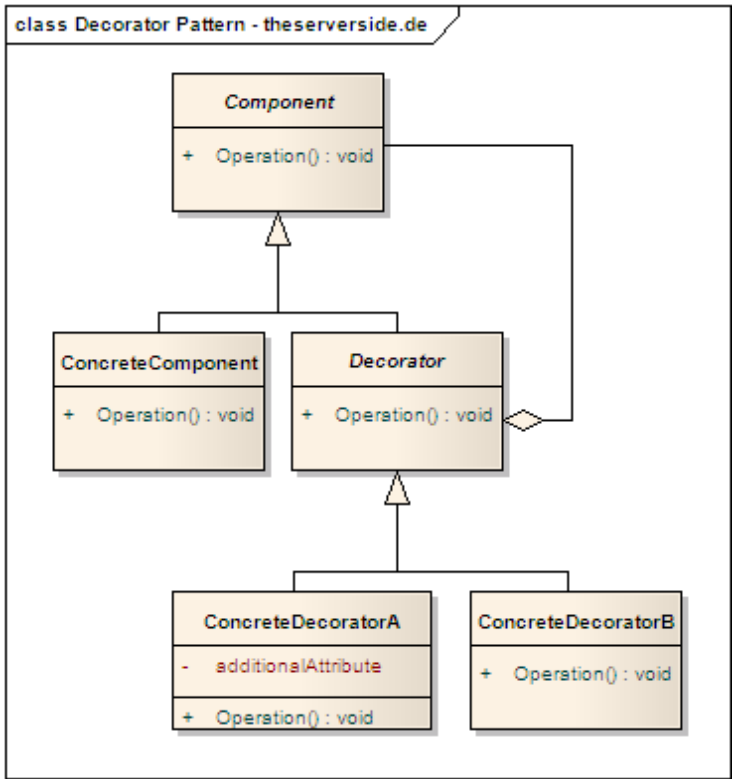


Abbildung [6]

In Python gibt es schon die Funktion „decorator“, deswegen muss man darauf Acht geben, dass man diese Funktion nicht mit dem „Decorator Pattern“ verwechselt.

Ein Python „decorator“ ist eine spezifische Änderung von der Python Syntax, welches uns somit erlaubt, dass wir bequem Funktionen beziehungsweise Methoden ändern können.

Hier ist ein kleines Beispiel zu der Funktion decorator:

Beispiel decorator

```
# decorator style
@decorator
def foo():
    pass

# "old" style
def foo():
    pass

foo = decorator(foo)
```

Abbildung [2] ergänztes Beispiel

Decorator Pattern Beispiel 1

```
class foo(object):
    def f1(self):
        print("original f1")

    def f2(self):
        print("original f2")

class foo_decorator(object):
    def __init__(self, decoratee):
        self._decoratee = decoratee

    def f1(self):
        print("decorated f1")
        self._decoratee.f1()

    def __getattr__(self, name):
        return getattr(self._decoratee, name)

u = foo()
v = foo_decorator(u)
v.f1()
v.f2()
```

Abbildung [3]

Wenn wir diese Klasse aufrufen, bekommen wir folgenden Output:

```
/Library/Frameworks/Python.framework/Versions/3.3/bin/python3.3 /Users/floriandienesch/PycharmProjects/untitled1/patterns/decorator/Decorator.py
decorated f1
original f1
original f2
```

Decorator Beispiel 2

```python
class Class(object):
    def __init__(self):
        pass

    def something_useful(self, string):
        return string

# Dekorierer
class Decorator(object):
    def __init__(self, wrapped):
        self.wrapped = wrapped

    def decorator1(self, string):
        return self.wrapped().something_useful(string).upper()

    def decorator2(self, string):
        string = '! '.join(string.split(' '))
        return self.wrapped().something_useful(string)


if __name__ == '__main__':
    string = 'Das ist ein Test :)'
    a = Class()
    print(a.something_useful(string))          # plain method
    b = Decorator(Class)
    print(b.decorator1(string))                # Decorator 1
    print(b.decorator2(string))                # Decorator 2
```

Abbildung [4]

Wenn wir diese Klasse aufrufen, bekommen wir folgenden Output:

```
/Library/Frameworks/Python.framework/Versions/3.3/bin/python3.3 /Users/floriandienesch/PycharmProjects/untitled1/patterns/decorator/Decorator2.py
Das ist ein Test :)
DAS IST EIN TEST :)
Das! ist! ein! Test! :)
```

Man kann sehen, dass der string durch den Decorator großgeschrieben wurde.

Quellen

[1] http://de.wikipedia.org/wiki/Strategie_(Entwurfsmuster), aufgerufen am 21.12.2014

[2] http://simeonfranklin.com/blog/2012/jul/1/python-decorators-in-12-steps/, aufgerufen am 20.12.2014

[3] http://stackoverflow.com/questions/3118929/implementing-the-decorator-pattern-in-python, aufgerufen am 21.12.2014

[4] http://stackoverflow.com/questions/8328824/whats-the-difference-between-python-decorators-and-decorator-pattern, aufgerufen am 21.12.2014

[5] http://www.theserverside.de/decorator-pattern-in-java/, aufgerufen am 22.12.2014

[6] http://www.theserverside.de/wp-content/uploads/2009/09/DecoratorPattern.png, aufgerufen am 22.12.2014